Conference or Workshop Item:
Swan, Jeremiah and Burles, Nathan John orcid.org/0000-0003-3030-1675 (2015) Hyper-quicksort: energy efficient sorting via the Templar framework for Template Method Hyper-heuristics. In: 39th CREST Open Workshop: Measuring, Testing and Optimising Computational Energy Consumption, 23-24 Feb 2015, UCL.

Hyper-quicksort: energy efficient sorting via the TEMPLAR framework for Template Method Hyper-heuristics

jerry.swan@cs.stir.ac.uk
nathan.burles@york.ac.uk

Motivation

Scalability remains an issue for program synthesis:

- We don't yet know how to generate sizeable algorithms from scratch.
- *Generative* approaches such as *GP* still work best at the scale of *expressions* (though some recent promising results [6]).
- **Formal** approaches require a strong *mathematical background*.
- ... but *human ingenuity* **already** provides a vast repertoire of *specialized algorithms*, usually with known asymptotic behaviour.

Given these limitations, how can we best use **generative hyper-heuristics** to **improve** upon **human-designed algorithms**?

The Template Method Pattern

- The **'Template Method' Design Pattern** [1] divides an algorithm into a *fixed skeleton* with one or more *variant* parts.
- The *fixed* parts *orchestrate the behaviour* of the *variant* parts.
- Example: Quicksort performance depends on the quality of the *pivot*, so we can treat the *pivot function* as a *variant part*:

```
DoubleArray qsort(DoubleArray arr) {
    double pivot = pivotFn( arr );
    // ^^^ pivotFn can be varied generatively
    return qsort( arr.filter( < pivot ) )
        ++ arr.filter( == pivot )
        ++ qsort( arr.filter( > pivot ) );
}
```

Template Method Hyper-heuristics [10]

- So if we can express an algorithmic framework in template method terms, then we can *learn good implementations* for the *variant parts*.
- By 'good', we mean 'biased towards the distribution to which the algorithm is exposed'.
- If our algorithms are *metaheuristics*, this means that they are *not subject to the 'No Free Lunch' theorem* [8], since the distribution over problem instances is *biased away from uniform* by the training set.
- Successfully demonstrated this approach to learn more effective GA selection and mutation operators [11, 9].

A framework for generative hyper-heuristics

Generative hyper-heuristics can be specified by:

- A list of **variation points** describing the parts of the algorithm to be automatically generated.
- An **algorithm template** expressing the algorithm skeleton. The template produces a *customized version of the algorithm* from *automatically-generated implementations* of the variation points.
- A **fitness function** to evaluate the customized algorithm.
- An **algorithm factory** that *searches the space of variation points* to produce an *optimized version of the algorithm*.

A functional description

For algorithm with function signature $I \rightarrow O$:

- $VP : (I_1 \rightarrow O_1) \times (I_2 \rightarrow O_2) \times \ldots \times (I_n \rightarrow O_n)$.
- Template : $VP \rightarrow (I \rightarrow O)$.
- Fitness : $(I \rightarrow O) \rightarrow V$.
- Factory : $VP \times Template \times Fitness \rightarrow (I \rightarrow O)$.

Why a Framework?

Generative HH are *laborious to implement* on a per-case basis, but *non-trivial to generalize*:

- The Factory is typically implemented via GP and is invoked repeatedly . . .
- . . . but popular GP implementations such as ECJ [3] and PushGP [7] *expect to be the 'top' of the system* . . .
- . . . hence are not easy to use for generative hyper-heuristics.
- Fitness of one VP depends on the other VPs, so *some fiddly software engineering is required* to enable 'dependency inversion'.
- *Heterogeneous signatures of VPs* needs special handling to retain any notion of type-safety.
- To prevent overfitting, cross-validation should be built-in to the fitness function by default.

Interlude - higher-order functions in Java

```
interface Fun1<Arg, Result> {
  Result apply(Arg arg);
}
interface Fun2<Arg1, Arg2, Result> {
  Result apply(Arg1 arg1, Arg2 arg2);
}

// We can then use functions as parameters
// and return values:
Fun1<Int, String>
compose(Fun1<Int, Double> f, Fun1<Double, String> g) {
    return new Fun1<Int, String>() {
        String apply(Int arg) {
            return g.apply( f.apply( arg ) );
        }
    };
}
```

Core TEMPLAR classes

```
public interface AlgTemplate<I,O> {
    public Fun1<I,O>
    makeAlg( ProgramList programs );
}

public class AlgFactory<I,O> {
    AlgFactory(GPConfig [] variationPointConfigs,
        AlgTemplate<I,O> template) { ... }

    ProgramList run(FitnessCases<I,O> cases,
        LossFn<O> lossFn) { ... }
}
```

Trivial example - 'Identity' template

Just executes the generated program for the (sole) variation point:

```
class IdentityTemplate
implements AlgTemplate<Double,Double> {

    public Fun1<Double,Double>
    makeAlg(ProgramList progs) {
        // Wrap the VP in a function:
        return new Fun1<Double, Double>() {
            Double apply(Double arg) {
                return progs.get(0).execute(arg);
            }
        };
    }
}
```

Using Templar

The end-user only needs to do this[1]:

```
// 1. Define an AlgTemplate subclass (previous slide).
// 2. Set up the algorithm-specifics:
AlgTemplate<Double,Double> template = new
    IdentityTemplate();
GPConfig[] vpConfigs={new RationalFunctionConfig();}
FitnessCases trainingSet = ...
FitnessCases testSet = ...

// 3. Invoke TEMPLAR:
ProgramList bestVPs = Templar.trainAndTest(template,
     vpConfigs,
     trainingSet,testSet,
    new RMSLossFn<Double>());
println("best VPs:" + bestVPs);
```

[1]These examples describe *all* the code you need to write.

Next simplest example - Composition Template

```
class CompositionTemplate
implements AlgTemplate<Int, String> {
    Fun1<Int, String> makeAlg(ProgramList progs) {
        f = new Fun1<Int, Double>() {
            Double apply(Int arg) {
                return progs.get(0).execute(arg);
            }
        };
        g = new Fun1<Double, String>() {
            String apply(Double arg) {
                return progs.get(1).execute(arg);
            }
        };
        // this template just composes
        // the two variant programs ...
        return compose(f, g);
    }
}
```

HyperQuicksort

- Just follow the above steps for *any* algorithm you wish to optimize.
- We'll see how easy it is to create 'Hyper-quicksort' ...

HyperQuicksort - Pivot Function

```
abstract class PivotFn
extends Fun2<DoubleArray, Intger, Double>{
  Double apply(DoubleArray a, Int recusionDepth);
}
class SedgewickPivotFn extends PivotFn {
  // counters the case of sorted
  // (or reverse-sorted) input
  Double apply(DoubleArray a, Int recusionDepth){
    return median(a.first, a[a.length/2], a.last);
  }
}

Int quicksort(DoubleArray a, PivotFn pivotFn);
// ^ instrumented to return some measure
// of pivotFn fitness (e.g. max recusion depth)
```

HyperQuicksort - Alg Template

```
class QuicksortTemplate
implements AlgTemplate<DoubleArray, Int> {
  Fun1<DoubleArray, Int> makeAlg(ProgramList progs) -> {
    PivotFn pivotFn = (DoubleArray a, Int recursionDepth)
        -> {
      int progResult = progs[0].execute(a.size,
          recursionDepth);
      int numSamples = min(abs(progResult),a.size);
      return median(randomSample(a,numSamples));
    };
    return (DoubleArray arg) -> Quicksort.sort(arg,
        pivotFn);
  }
}
```

HyperQuicksort - Top Level

```
// 1. Define an AlgTemplate subclass (previous slide).
// 2. Configure GP to generate pivotFn VP:
List<Var> vars = {Var("size"),Var("recursionDepth");
List<Node> funcSet = {IfFn(),LessFn(),AddFn(),...};
GPParams params = ... // crossover, selection etc
GPConfig vpConfigs={new GPConfig(funcSet,vars,params);}

// 3. Invoke TEMPLAR
AlgTemplate<Double,Double> template = new
    QuicksortTemplate();
FitnessCases trainingSet = ...
FitnessCases testSet = ...
Templar.trainAndTest(template,vpConfigs,trainingSet,
    testSet,new RMSLossFn<Double>());
```

Wait - there's more . . .

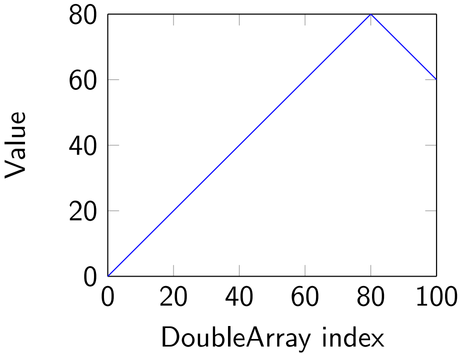
- Manual creation of GP nodes for function sets on custom solution representations (e.g. Timetable,RoutePlan,AntTrail etc) is tedious.
- Following [2], Templar.FunctionSetGenerator uses reflection to **automatically build a function set** from *any* Java object.
- By this means, a hyper-heuristic for *Iterated Local Search over bitstrings* was up and running from scratch **in under 20 minutes**
- By following the above steps, it's quick and easy to create a template for **your favourite algorithm here**.
- All you need now is *lots of CPU time* . . .

Experiment - Setup

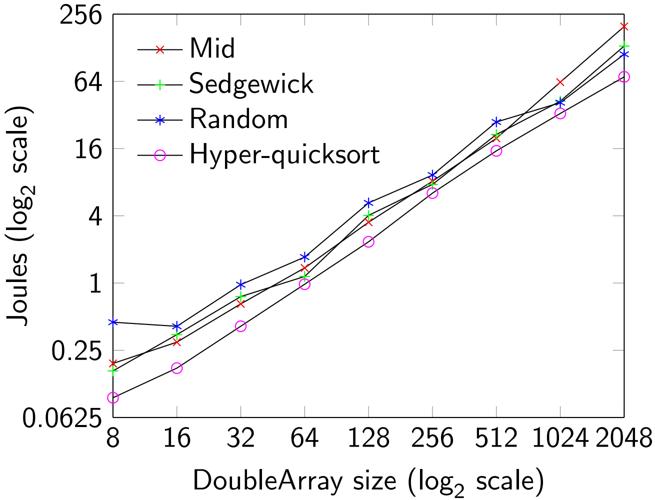
- EpochX for GP.
- JALEN [5] for power measurement.
  - Monitors execution time and processor utilisation to estimate power consumption.
  - Non-deterministic (e.g. other processes), and accuracy limited by platform (up to nanosecond).
  - Multiple arrays need to be sorted for each measurement (100).
  - Oracular pivot function.

Experiment - Pipeorgan Distribution [4]

Training set size: 70 (* 100).
Testing set size: 100 (at 9 different array lengths, * 1000).



Results


Mid
Sedgewick
Random
Hyper-quicksort
Joules ($\log_2$ scale)
256
64
16
4
1
0.25
0.0625
8
16
32
64
128
256
512
1024
2048
DoubleArray size ($\log_2$ scale)

Results - Table

Array size	Middle index J	Middle index p	Middle index e	Sedgewick J	Sedgewick p	Sedgewick e	Hyper-quicksort (J)
8	0.191	7.46e-32	0.981	0.163	8.37e-32	0.980	0.094
16	0.296	1.20e-30	0.971	0.345	1.25e-31	0.979	0.173
32	0.651	8.13e-32	0.980	0.757	7.25e-32	0.981	0.410
64	1.366	4.80e-33	0.990	1.145	1.68e-30	0.970	0.976
128	3.505	4.80e-33	0.990	4.034	4.14e-33	0.991	2.341
256	8.175	4.14e-33	0.991	7.646	3.41e-32	0.983	6.387
512	19.777	4.33e-34	0.998	21.391	3.62e-34	0.999	15.268
1024	62.961	2.52e-34	1.000	42.508	6.44e-33	0.989	33.012
2048	198.438	2.52e-34	1.000	132.663	2.52e-34	1.000	70.234

Array size	Random index J	Random index p	Random index e
8	0.446	8.37e-32	0.980
16	0.410	8.37e-32	0.980
32	0.967	4.80e-33	0.990
64	1.708	4.80e-33	0.990
128	5.221	2.52e-34	1.000
256	9.269	8.87e-34	0.996
512	27.685	2.52e-34	1.000
1024	41.245	3.61e-32	0.983
2048	111.894	3.47e-33	0.991

Experiment - Conclusions

- P-values (Mann-Whitney U-test) and effect sizes (Vargha-Delaney $\hat{A}_{12}$) clearly show Hyper-quicksort provides significant improvement on pipeorgan distributions.
- Intermediate results showed that minimal recursion doesn't always equate to minimal power consumption, as pivot function becomes more demanding.
- Imprecision and non-determinism of power measurement imposes time constraints on experimentation.

Conclusion and Future Work

- Algorithms can be decomposed into *templates* consisting of a fixed skeleton and a collection of variant components.
- By judicious choice of function signatures, we can use generative methods (GP etc) to create variant components that are tuned to some target distribution.
- In implementation terms, TEMPLAR makes **generative HH for any algorithm** a matter of **GP parameter tuning**.
- New methods of power consumption modelling are in development. . .

References I

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: elements of reusable object-oriented software.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Simon M. Lucas.
Exploiting reflection in object oriented genetic programming.
In Maarten Keijzer, Una-May O'Reilly, Simon Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming*, volume 3003 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin Heidelberg, 2004.

References II

Sean Luke, Liviu Panait, Gabriel Balan, and Et.
ECJ 16: A Java-based Evolutionary Computation Research System, 2007.

M. Douglas McIlroy.
A killer adversary for quicksort.
Softw., Pract. Exper., 29(4):341–344, 1999.

Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier.
Runtime monitoring of software energy hotspots.
In *27th IEEE/ACM Int. Conf. on Autom. Softw. Eng. 2012*, pages 160–169. IEEE, 2012.

References III

Lee Spector, Kyle Harrington, and Thomas Helmuth.
Tag-based modularity in tree-based genetic programming.
In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 815–822, New York, NY, USA, 2012. ACM.

Lee Spector, Jon Klein, and Maarten Keijzer.
The Push3 execution stack and the evolution of control.
In Hans-Georg Beyer et al, editor, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

References IV

John Woodward and Jerry Swan.
Why classifying search algorithms is essential.
In *2010 International Conference on Progress in Informatics and Computing. (PIC-2010)*, 2010.

John R. Woodward and Jerry Swan.
The automatic generation of mutation operators for genetic algorithms.
In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 67–74, New York, NY, USA, 2012. ACM.

References V

- John R. Woodward and Jerry Swan.
  Template method hyper-heuristics.
  In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1437–1438, New York, NY, USA, 2014. ACM.

- John Robert Woodward and Jerry Swan.
  Automatically designing selection heuristics.
  In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 583–590, New York, NY, USA, 2011. ACM.